Proceedings, Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan, August 1979.

(6)

# A KNOWLEDGE-BASED APPROACH TO USING EFFICIENCY ESTIMATION IN PROGRAM SYNTHESIS

(10) Elaine Kant[*]
Artificial Intelligence Laboratory
Stanford University
Stanford, California 94305

(15) N00014-79-C-0127, W DARPA Order-3687 (11) 1979 (12) 7p.

This paper describes a system for using efficiency knowledge in program synthesis. The system, called LIBRA, uses a combination of knowledge-based rules and algebraic cost estimates to compare potential program implementations. Efficiency knowledge is used to control the selection of algorithm and data structure implementations and the application of optimizing transformations. Prototypes of programming constructs and of cost estimation techniques are used to simplify the efficiency analysis process and to assist in the acquisition of efficiency knowledge associated with new coding knowledge. LIBRA has been used to *guide the selection of implementations for several programs that classify, retrieve information, sort, and generate prime numbers.*

## 1. INTRODUCTION

Efficiency considerations often impose conflicting demands on a program synthesis system. On the one hand, a synthesis system must produce an efficient target language program; on the other, it must produce that target code in a reasonable amount of time and without running out of storage. This paper discusses a system that takes a middle ground between the extremes of 1) constructing all possible programs that meet the specification and picking the most efficient, and 2) using default implementations. The system, called LIBRA, uses a knowledge base of *efficiency rules* to guide the construction of *relatively efficient* target language programs in a reasonable amount of time. LIBRA works from a more abstract specification and considers a wider range of target-language implementations than optimizing compilers. Many choices must be made, and making a good decision depends on a global view of the program. The target programs are not guaranteed to be optimal, but the efficiency knowledge is designed to allow the flexibility of trading off target-program efficiency for speed and compactness in the synthesis process.

The basic paradigm is heuristic search through a set of more and more complete program descriptions. Estimates of the execution costs of program implementations are used as evaluation functions in the search. Symbolic, algebraic program analysis is used to estimate the execution costs. Knowledge about the time and storage costs of data structures and operations is used to choose combinations of algorithms and data representations and to control the application of optimizing transformations. Rules about plausible implementations are used to prune the search tree. LIBRA has been been used to guide the construction of several variants of programs that retrieve information, sort, classify, and generate prime numbers.

## 2. BACKGROUND

LIBRA is an extension of an interactive program synthesis system *that generates implementations in a target language by a series* of transformations and refinements of program descriptions, called *coding rules*. The knowledge base of coding rules was *developed by Barstow [1]. The knowledge base allows programs* in the area of symbolic processing to be specified in terms of constructs including sets, mappings, set operations, and enumeration. The knowledge in both the coding rules and efficiency rules permits the construction of programs using lists, arrays, hash tables, property lists, and several enumeration, sorting, and searching constructs. The target programs are written in a subset of INTERLISP.

Most of the rules are not specific to the target language. For example, there are 5 or 10 rules that gradually refine a set into a hash table, and then a few language specific rules for refining the hash table into LISP. Although the general paradigm is refinement from abstract to more detailed program *descriptions*, transformations such as combining nested blocks of code or nested loops are also allowed. LIBRA decides whether or not to apply such a transformation just as it decides which of several refinements to apply, by looking at the global execution cost estimates or by applying heuristics.

LIBRA and the coding rules function together both as the synthesis phase of the PSI program synthesis system [2] and as an independent synthesis system. Figure 1 shows a simplified *view of the synthesis phase* and its relation to the rest of PSI. The other modules of the PSI system allow the description of programs by English dialogue or by examples or traces, and translate the specification into a complete high-level language description. A specification in this high level language can also be given directly to the synthesis phase.

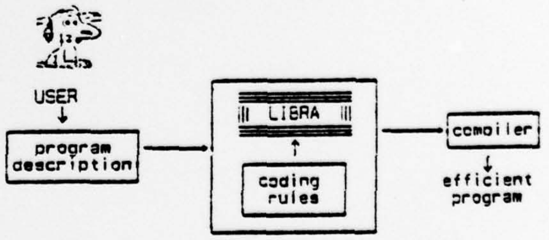


Figure 1. A LIBRA's eye view of program synthesis in PSI

## 3. PREVIEW

LIBRA chooses from among applicable refinements in the knowledge base of coding rules through additional sets of rules that can be easily modified. For example, rules about planning, derived from previous analyses of how to make particular implementation decisions, reduce the effort of explicitly constructing and comparing alternative implementations. Related decisions are grouped to reduce the size of the search space and to make cost tradeoffs more obvious. Rules about scheduling and resource allocation set priorities that reflect the importance of a coding decision and the effort expended in making the choice.

* This research was supported in part by a Fannie and John Hertz Foundation Fellowship, in part by a National Science Foundation Fellowship, in part by Systems Control, Inc., under the Defense Advanced Research Projects Agency Order 3687, Contract N00014-79-C-0127, and in part by the Stanford Artificial Intelligence Laboratory under the Defense Advanced Research Projects Agency Order 2494, Contract MDA 903-76-C-0206.

389 333 LB

When appropriate, alternate implementations are explicitly constructed and compared analytically. The comparisons use global cost estimates to reflect the interdependence of decisions. The cost estimations can be made at any stage of the refinement process, although estimates of more completely refined programs are generally more accurate. LIBRA computes upper and lower bounds on the estimated execution cost and uses them for pruning program implementations with branch and bound. These bounds are also useful in identifying parts of the program that might lead to bottlenecks. Refinement resources are then concentrated on those parts of the program.

Since the knowledge-based is modular, it facilitates the acquisition of new programming knowledge. The same prototypes of programming constructs and of cost estimation procedures that simplify the efficiency analysis process are also quite useful in adding the efficiency information to match the coding knowledge that is in the system. A semi-automated process for adding new efficiency information has been developed.

The focus of this article is on the overall efficiency framework and on the knowledge-based aspects of LIBRA. More details on the analysis procedures and on other topics only covered briefly here can be found in [3].

## 4. THE PROBLEM

The question addressed here is how to select an efficient implementation for a high-level program specification, given a set of rules for constructing the possible implementations. It is assumed that there may be a very large number of possible implementations and that it is not possible to construct and compare all possibilities explicitly. The design goal was to produce a system that would *automatically* select implementations and that would be compatible with the refinement paradigm for program synthesis.

The following example illustrates the type of problem that LIBRA solves. The problem is to synthesize a good implementation of a simple database retrieval program.

> The program first inputs a database of news stories. It then loops, accepting a keyword command and printing a list of all stories in the database that contain that keyword, alphabetized by story name. The special keyword "xyzzy" causes the program to terminate.

As part of the program specification, the user may specify information such as the estimated number of times a keyword command will be given, the expected number of stories in the database, and the average number of keywords per story. Some variations of this example are developed further in later sections.

### 4.1 Implementation issues

Given a high level program description, there are several types of implementation issues to be considered:

-- choosing data structure representations
-- implementing high level operations
-- applying optimizing transformations

Some of the major difficulties in resolving these issues arise from the need to consider:

-- time and space trade-offs
-- dependencies among decisions
-- efficiency of target program versus
efficiency of synthesis

Thus, in the program described above, a representation for the database must be chosen, and a method for finding the stories associated with the keyword must be chosen. If there is the an opportunity to apply a transformation such as combining two loops, it must be determined whether that transformation will actually improve the performance of the target program.

Often there is no ideal representation that minimizes both space and time. In the news retrieval example, the database can be represented as a mapping from stories to sets of keywords. Unless the database is relatively small, it will take quite some time to search for all the stories containing the given keyword and to sort that list. Another possibility is to use an additional representation of mappings from keywords to a sorted list of stories containing that keyword. If keyword searches are requested frequently, this would improve the running speed, but at the expense of additional storage space.

When more than one data structure is involved, it may not be possible to make implementation decisions independently. Given most cost functions, there will be cross-product terms involving the space from one representation and the time from an operation on another. For example, this could happen if the cost function were the product of 1) execution time of a statement, 2) number executions, and 3) total storage in use, summed over all statements in the program. These cross-product terms make it impossible to analyze the costs of the decisions independently. The best implementation choice also depends on the relative frequency of the retrieval operations and the sizes of the data structures.

### 4.2 Some subproblems

Some subtasks of this general problem of finding an efficient implementation include codifying the efficiency knowledge needed to:

**1) symbolically estimate and compare execution costs**
One way to choose a good implementation is to make several alternative refinements, estimate the costs of the resulting program implementations, and choose the best one.

**2) store and apply previous efficiency analysis results**
To avoid excessive analysis, it is helpful to be able to exploit the results of previous analyses. So there should be a mechanism for adding rules such as:

> *"In refining a set that has more than 30 elements and that is used only to test membership and add and delete elements, the hash-table representation is a good choice."*
>
> "In refining a sequentially represented set in which elements are frequently inserted and deleted, use a linked list rather than an array." (This avoids shifting.)

**3) concentrate effort on important parts of the program**
The synthesis system should determine whether the representation of the database has a greater effect on the global program cost than the choice of alphabetizing technique, and should use that information to focus synthesis resources.

### 4.3 Related research

Only some of the types of efficiency knowledge described in the previous section have been codified for machine use. The primary research has been in data-structure selection systems. Some verification and theorem proving systems can prove facts about the execution performance of programs, but they do not use this information to guide program synthesis. The use of efficiency knowledge in program synthesis has not been addressed by debugging or analogy approaches.

The data-structure selection systems all use cost estimation for comparison of implementations. Low [4] uses numerical cost estimates to choose data structures from among a library of implementations. To find branching probabilities, the system inserts statement counts into a default implementation that is run on sample data. Set sizes at different points in the program are determined by querying the user. Morgenstern's system, a part of PROTOSYSTEM-I, [5], uses estimates of file input/output and sorting costs to choose file system organizations and order the flow of processing operations in management information system.

These systems include heuristics for avoiding complete search, but the heuristics are not always expressed explicitly. Low's system has a built-in rule for avoiding multiple representations by forcing all data structures to have the same representation throughout the program and by constraining all data structures that are arguments to a common operation to share an identical representation. Rovner [6] extended Low's work to the selection of associative data structures and also allowed the selection of redundant representations. Heuristics about when to consider redundant representations and about other cost-tradeoff assumptions were carefully noted in the description of the system, but were not expressed as independent rules in the system implementation.

Several different search strategies have been tested. Low and Rovner use hill climbing among the estimated costs of the target programs to choose an implementation. Morgenstern uses a dynamic programming algorithm specifically tailored to choose structures for large files. Wegbreit [7] gives some examples of the use of performance analysis to drive a program transformation process. LIBRA represents its resource-management strategy in rules. One of the rules, which suggests consideration of the high potential impact decisions first, is similar to the techniques used by Wegbreit and Morgenstern.

Several other approaches to the problem of data structure selection have been taken. The SETL project [8] uses a more traditional optimizing compiler approach to choose set representations based on a small set of alternatives. The systems described in [9] and [10] attempt to match modelling structures with the user's needs. An unsolved problem in this approach is how to combine several modelling structures into one representation.

## 5. A FRAMEWORK FOR EFFICIENCY ESTIMATION

LIBRA was designed to explore the feasibility of combining analytic and knowledge-based approaches to efficiency estimation. The basic idea in the framework is heuristic search through a tree of partially implemented program descriptions. Efficiency rules from LIBRA are used to control the search and to add efficiency-analysis information to the program description. Coding rules from Barstow's knowledge base are used to refine the program description into a more concrete description.

The root node of the search tree is the initial program specification and the leaf nodes are target language programs. Each of the intermediate nodes is a partially implemented version of the entire program. The order in which refinements are considered affects the subtree that is constructed. The focus of attention for refinement may be limited to a particular part of the program, but comparisons between nodes are based on global execution costs. The tree of partial program implementations, each with an agenda of synthesis tasks, serves as a workspace for recording the state of the search (see Figure 2 below).

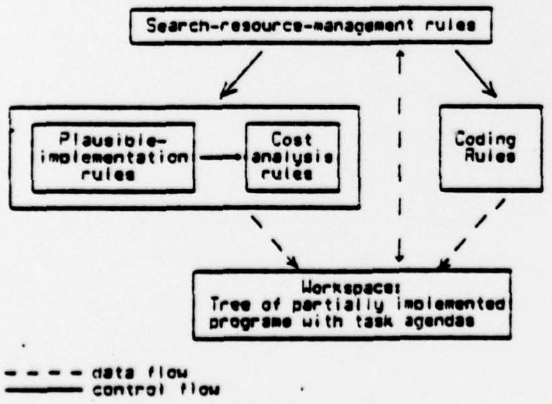

Search-resource-management rules
Plausible-implementation rules
Cost analysis rules
Coding Rules
Workspace: Tree of partially implemented programs with task agendas
- - - - data flow
——— control flow

**Figure 2. Overview of efficiency framework.**

A somewhat simplified description of the search strategy is: pick a program implementation to work on, pick a refinement task within that implementation, pick a coding rule to achieve that task, and finally apply the coding rule and any associated efficiency rules.

*Search-resource-management* rules choose a program implementation and then a part of that program to work on. These rules assign priorities to tasks to ensure that the tasks are carried out within the limits of the resources.

When refining a part of a program, all relevant coding rules are retrieved and tested for applicability. *Plausible-implementation rules* are used to help decide which coding rule to apply. These rules contain precomputed analyses and are used to restrict the possible coding rules to those that seem reasonable in the given program situation, thus pruning the search tree.

Sometimes several coding rules seem plausible. Separate program descriptions are set up and refined, then compared using the cost estimates determined by *cost-analysis rules*. Search-resource-management and plausible-implementation rules may call on the cost-analysis rules for symbolic execution cost estimates to compare different implementations and identify potential bottlenecks in the target program execution.

### 5.1 Assigning priorities to decisions

Since all implementations cannot be considered in equal detail, the quality of the decisions depends on the order in which they are considered and the depth to which the consequences are explored before making a commitment. The search-resource-management rules use scheduling and resource allocation to balance the final program performance with the cost of choosing and constructing the implementations.

*Task-ordering rules* determine the ordering for attempting different refinement tasks. Ordering principles include expanding complex programming constructs, such as "SUBSET" early to expose choices, and postponing choices of refinement rules and low level coding details until the major decisions have been made.

*Choice-ordering rules* find an order for considering the decisions that must be made. One of these rules suggests allocating the most resources to the decisions that are likely to lead to bottlenecks and making those decisions first. Section 5.3 describes how these *high potential impact* decisions are identified. LIBRA makes an adjustment to the potential impact of a decision to reflect the accuracy of cost estimates for the current level of program development and the expected cost of completing the refinement process. Without this, a highly refined implementation might be abandoned in favor of a very abstract description with a slightly better optimistic estimate that is probably not achievable.

### 5.2 Applying plausible-implementation rules

The plausible-implementation rules in LIBRA describe the situations under which data structure implementations are appropriate, when different sorting operations are plausible, and when to consider using more than one representation for a data structure. This knowledge is used to compare implementations without the expense of explicit construction and evaluation of execution costs of all alternatives.

The plausible-implementation rules are structured condition-action rules. The condition of a rule about data structures, for example, states all the critical uses of a data structure that make the rule relevant. Efficiency information such as the size of a data structure and the number of executions of a statement may be used in the rule condition. The rule action can set a Boolean combination of constraints for a set of program parts requiring that they be refined (or not refined) to a particular programming construct. A three valued logic (satisfied, impossible, possible) is used to check constraints.

### 5.3 Estimating execution costs

LIBRA includes a knowledge base of rules for estimating the execution cost of a program description at any stage of the refinement process and with varying degrees of accuracy. The user is expected to provide some basic information about the program, and then LIBRA keeps the analysis updated for the rest of the refinement process. For example, in the NEWS program, the basic information needed is the expected number of stories, the average number of keywords per story, and the number of times the main loop in the program will be executed for a given database. LIBRA then makes analysis transformations in parallel with refinements so that more accurate cost estimates can be associated with succeeding nodes in the tree. Some analysis rules are associated with particular coding transformations. Many rules, such as those for analyzing Boolean combinations, are associated with coding constructs rather than transformations. Information about parameters such as data structure sizes, statement running times and execution frequencies, and data structure usage information is maintained.

The top-down, incremental analysis allows programs to be analyzed that would be difficult to analyze automatically if only the target program were presented. An advantage of combining the stepwise refinement with this sort of analysis is that classes of implementations can be compared by considering the cost estimates for intermediate program descriptions rather than explicitly expanding the tree and comparing the target language programs.

Estimating execution costs is not an exact science. LIBRA attacks the problem by using both upper and lower bounds on the execution cost. The upper bound, or *achievable* estimate, is calculated by introducing a *standard implementation* for each of the programming constructs used and by assuming that standard implementation choices are made for the rest of the refinement process. The lower bound, or *optimistic* cost estimate is based on a lower bound for implementations known to the program, not a theoretical lower bound. Global optimistic cost estimates are estimated by assuming optimistic costs for each of the constructs in the program and by assuming that no representation conflicts occur.

The importance of a decision is measured by its *potential impact*. This is achievable bound cost estimate and the execution cost estimated when optimistic cost estimates are used for all parts of the program involved in the decision.

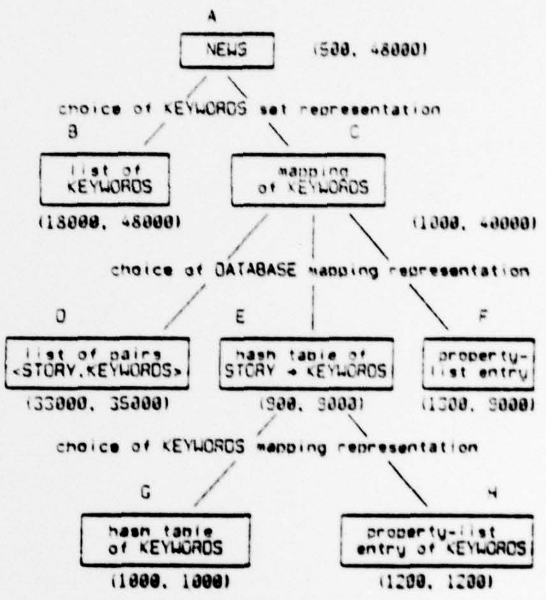
A general model of program constructs and specific models for each construct are used to organize the cost estimation process. Also, a standard cost-computation process allows sharing of subroutines between estimation strategies for making quick estimates and for performing more detailed (and usually more expensive) analysis.

## 6. AN EXAMPLE

This section will consider the implementation of a retrieval program in more detail. The problem to be implemented, called NEWS, is:

> Read in a database of news stories. The DATABASE is a mapping from stories to sets of KEYWORDS. Repeatedly accept a keyword and prints out a list of the names of the stories in the database that contain that keyword. When the special command "xyzzy" is given instead of a keyword, then halt.

LIBRA has directed the implementation of several versions of NEWS. Under different assumptions about the size of the database or the cost function to be used, different implementations are selected. Figure 3 below shows the tree of implementations that is generated and searched under certain assumptions about data structure sizes and branch probabilities. The major choices to be made in implementing NEWS are choosing representations for the DATABASE mapping and for the KEYWORDS set.



A
NEWS
(500, 48000)
choice of KEYWORDS set representation
B
list of KEYWORDS
(18000, 48000)
C
mapping of KEYWORDS
(1000, 40000)
choice of DATABASE mapping representation
D
list of pairs <STORY,KEYWORDS>
(33000, 35000)
E
hash table of STORY → KEYWORDS
(900, 9000)
F
property-list entry
(1300, 9000)
choice of KEYWORDS mapping representation
G
hash table of KEYWORDS
(1000, 1000)
H
property-list entry of KEYWORDS
(1200, 1200)

**Figure 3. Overview of NEWS implementation.**

### 6.1 Alternate implementation paths

A number of ways to implement NEWS are possible with the current set of coding rules. One refinement path, node G in the search tree of Figure 3, is followed through in more detail in the following sections. It involves representing DATABASE internally as a hash table of stories, with each story in turn having a hash table of keywords. The cost function used in this case is the product of running time and number of pages in use. LIBRA chooses a hash-table representation for KEYWORDS because there are many keywords for each story. The time to convert the set of keywords into a hash table is balanced by the time savings from the membership test, which is faster as a hash-table look-up than as a search through the list of keywords (for large keyword sets). The DATABASE representation decision is similar. Both choices are reinforced by the fact that the main loop is executed many times before exiting with "xyzzy."

Under other assumptions, a path through node B is taken and a linked-list representation is selected. If the loop is executed only a few times or if the number of keywords associated with a story is small, then the time required to convert the database from the list of pairs (<story, keywords>) representation to a hash-table representation is not outweighed by the fast hash-table look-up operations. If space is a critical factor in the cost function, another path through B is taken in which the original representation of a list of pairs is preserved. This avoids using any additional space, but at a cost in time.

A different tree than the one pictured in Figure 3 may also be searched. Suppose there are only a few keywords per story, many stories, and a cost function dominated by running time. Then the representation of the DATABASE mapping is a more critical decision than the KEYWORDS set representation, because the time for the membership test would not differ greatly for the different representations. If fewer resources are available for synthesis than in the examples described above, then some of the less reliable plausible-implementation rules are used. For example, nodes F and H are not considered when a plausible-implementation rule that prefers hash-table representations to property-list entries is applied.

The implementations that LIBRA chooses in this case are about the best possible with the current set of coding rules. People can do better on the NEWS example by using representations outside the scope of the coding rules. However, for any given set of coding rules, allowing people to make the decisions would not produce better implementations.

### 6.2 Initial refinements in NEWS

The following sections show more details of the path leading to node G. By questioning the user, LIBRA determines that the expected number of stories in the database is 80, the average number of keywords per story is 100, the expected number of iterations of the loop is 300, and the probability that the command is a keyword of the average story is .01.

LIBRA first calls on the coding rules to make refinements that do not involve any decisions. For example, the input DATABASE is refined to the standard input format for mappings, a list of pairs <story, keywords>, and the set of KEYWORDS is refined into a linked list. LIBRA applies plausible-implementation rules to decide whether to consider multiple representations for the KEYWORDS set and the DATABASE mapping.

During refinement, a "for-all" statement enumerating the domain of DATABASE is created. It is refined into an explicit enumeration of the items of domain, since only one coding rule is applicable. To decide how to refine the enumeration, more information about the representation of the domain is needed. LIBRA does not consider all possible representation of the domain set explicitly; the choice is made by the application of plausible-implementation rules. For example, two of the efficiency rules about sets are:

> If the only uses of a set A are for enumerations over that set, and if B is another representation for A that is easily enumerable, then use the same representation for A as for B.
>
> If all uses of a set are for enumerations, or as pointers to positions in set, or as tests of the state of the enumerations, and if the target language is LISP, then refine the set into a linked list.

These rules determine that domain set, which is used only for enumeration and is not an alternate representation of some other set, should be refined into a linked list. a linked-list should be used. Therefore constraints on the domain set are established, and it is refined into a sequence, and then into a list (rather than an array) with the choices between applicable coding rules resolved by the constraints.

Some of the details of constructing the domain list and the enumeration of the domain are postponed by search-resource-management rules because LIBRA predicts that no decisions will be involved and the cost estimate for that part of the program will not change significantly. Other choices that arise and cannot be resolved by plausible-implementation rules are also postponed until other useful refinements are finished.

### 6.3 Identifying the most important decision

All of the changes above take place in node A of Figure 3. During this refinement, several choices are postponed. These choices are 1) how to refine the DATABASE mapping used inside the for-all, and 2) how to refine the KEYWORDS set within that mapping. What is the effect of each of the two choices to be made in this example?

The internal representation of DATABASE, (DB1), is used for retrieving the map value (keyword sets) of stories once per story per command. Possible implementations for mappings range from a linked-list format that make retrieval linear in the number of stories to associative structures that have nearly constant retrieval time.

The keyword sets in DB1 (KEYWORDS1), are used in a "member(command, KEYWORDS)" test. This test is executed one for each story for each iteration of the loop. Possible implementations give membership tests with times ranging from linear in the number of keywords to nearly constant.

Since the number of keywords is greater than the number of stories, the keyword representation has the largest cost differential and is more likely to be a bottleneck in the final program if care is not taken in the representation choice. According to the choice-ordering rule about making high potential impact decisions first, the next step is to look at the possible refinements of KEYWORDS1.

Decision-making resources are assigned. Currently the resources measured are the CPU time used in carrying out the refinements and the number of nodes used in the refinement trees. The resources needed to complete a program implementation without making choices are estimated and subtracted from the total available resources. Decision-making resources from the remainder are assigned in proportion to the estimated importance of the decision. Then, separate program descriptions are set up (actually they share some substructure) in which each of the alternate coding rules are applied. In this decision, the applicable rules allow either refining the keyword set into an explicit set, leading to search node B, or into an explicit mapping, leading to search node C.

### 6.4 Exploring two implementations for KEYWORDS1

LIBRA's goal is to refine the alternatives (B and C) enough so that the comparison among implementations will be informative. The resources previously assigned give upper limits on the time and space to be spent on getting a more accurate estimate of the program cost of the implementation being explored. Each program description also has a "purpose" to be fulfilled, which serves as a test of whether the task has been achieved and is used to set some of the task and choice-ordering strategies. There is also a set of program parts that is to be the focus of attention of processing. In this case, the KEYWORDS1 data structure and the representation conversion and the membership test are included in the focus set.

In the first program description, search node B, the explicit-set rule is applied and refinement proceeds until all relevant tasks are satisfied -- the resources allowed for writing the program are generous in this example. At the conclusion, the keyword set for each story has been refined, after the application of several coding rules, into a LISP list, and the membership operation has been refined into a list search.

Refinement of search node C, the program description in which the explicit-mapping rule was applied, also halts because all relevant tasks have been accomplished. Here the keyword set is refined to a mapping and membership tested by seeing if there is mapping for the given key. There is also a representation conversion since the keyword set is represented as a list in the input.

LIBRA then computes optimistic and achievable bounds on the cost of the whole program for each program description. In the linked-list implementation, B, the optimistic estimate is 18000 millisecond-pages, and the achievable bound is 48000. The optimistic and achievable cost estimates for the mapping representation, C, are 1000 and 40000 respectively. Branch and bound is applied to eliminate any implementations with optimistic estimates worse than the achievable estimate of some other implementation. Neither implementation is eliminated in this case, though later in the refinement of NEWS this technique will be fruitful. Node C has the best optimistic estimate and is chosen for further refinement.

### 6.5 Refining the rest of NEWS

The remaining decisions are choosing a refinement for the explicit mapping of KEYWORDS1 and choosing a refinement for DB1. The database decision is chosen by the potential impact method. Three program descriptions are set up to consider the three applicable refinement rules -- one to consider refining the mapping to a list of pairs (search node D), one to consider a stored mapping (node E), and one to consider a distributed mapping (node F). The relevant parts of the program, those related to the DB1 decision, are then refined in each program description. For example, the stored mapping is refined to a hash table. The resulting program descriptions are then compared with each other and with other program descriptions that have been temporarily abandoned, such as the search node B. As Figure 3 shows, nodes B and D can be eliminated from further consideration because even their lower bounds are worse than achievable bound on node E. The most promising implementation, search node E, is then chosen and refinement continues.

The final decision to be made is how to represent the KEYWORDS1 set, which has been refined into a mapping. As in the refinement of node C, there are three applicable coding rules. However, there is an applicable plausible-implementation rule about mappings that eliminates one of the possibilities.

> If a mapping has already been refined from a set, then do not refine it into a set of pairs.

Thus, only two coding rules are considered. These rules are both tested, in search nodes G and H. The stored mapping, leading to the hash table representation in node G proves to be the best choice. At this point, the cost estimate is precise enough to eliminate all the other possibilities. Thus, the best possibility is the implementation of both the keyword set and the mapping DB1 as hash tables. As refinement continues, several other choices of coding rules are presented, but they are all resolved by plausible-implementation rules. The decisions made include choosing to recompute rather than store values that are easy to compute. The program description is finally refined into a LISP program.

## 7. KNOWLEDGE ACQUISITION AIDS

LIBRA includes mechanisms to assist in the acquisition of new programming constructs, including the additions that are made to efficiency knowledge when new coding knowledge is added. When new high-level constructs are added, such as new types of sorts, or trees, new efficiency knowledge is needed to analyze these constructs, their subparts, running times, and other efficiency properties. LIBRA's prototypes of programming constructs are consulted by acquisition-aid routines when new constructs are added. Some of the necessary information can be deduced automatically, and the user is asked specific questions to obtain the rest.

Estimates of running time and space usage depend on the target language and target computer. LIBRA provides a semi-automatic procedure for deriving cost estimation functions from the set of functions for the target language constructs. This procedure can be used in to update efficiency rules when new coding rules are added. Currently only times estimating functions are derived, but a similar process could be used to check the accuracy of the plausible-implementation rules in the system when new coding knowledge is added.

## 8. CONCLUSIONS AND FUTURE DIRECTIONS

The use of efficiency estimation in program synthesis is a new but promising field. The issue of data-structure selection has been studied in some detail, but not the issue of estimating the effects of applying high level program transformations. LIBRA provides a framework in which both data-structure and algorithm selection can be treated. The heuristics that suggest orderings for considering refinement tasks and decisions and that suggest plausible implementations and when to consider multiple implementations are expressed explicitly as rules. A start has been made on symbolic algorithm analysis, and incremental analysis is used to make the analysis process tractable. One of the goals in LIBRA is to break up the programming process into manageable chunks in order to learn more about the sequences of implementation choices available, how the choices interact, and when and how the choices should be made.

To extend LIBRA to complete automatic programming system, additional research would be needed. For example, to write more complex programs such as compilers or operating systems, more coding and efficiency rules about constructs such as bit-packing, machine interrupts, and multiprocessing would need to be added to the system. However, the efficiency techniques described here should be sufficient to control combinatorial explosion.

Higher level optimizations, extended symbolic analysis and comparison capabilities, and more domain expertise are some feasible extensions to LIBRA. Another possibility is to automate the checking of conditions in the heuristic rules by doing a complete search through the current set of coding rules. Automatic generation of heuristics based on analysis of symbolic cost estimates would be another important addition. Adding an inference process to both the coding and efficiency estimation process would also be useful, though not as straightforward.

More powerful symbolic comparison techniques are also possible. For example, the range of values for which one implementation dominates another ($c1 \bullet N^2$ over $c2 \bullet N$) could be determined. The user would then only have to say whether N was within a particular range, rather than giving a definite value. Another use of symbolic costs is in proposing alternate solutions, each with the conditions that make that solution the best choice. If, for example, the cost for primitive operations such as multiply are given as ranges, the system could produce the solution "implementation X is best if the target machine has a very fast multiply, but implementation Y is best if multiplication takes about the same time as addition."

LIBRA has demonstrated the feasibility of the approach described here, but has by no means exhausted the research topics in efficiency estimation for program synthesis.

## REFERENCES

[1] Barstow, D. R. **Knowledge-based Program Construction.** Elsevier North-Holland, New York, 1979.

[2] Green, C. C. "The Design of the PSI Program Synthesis System." in *Proceedings of the Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976, 4-18.

[3] Kant, E. *Efficiency Considerations in Program Synthesis: A Knowledge-based Approach.* Forthcoming Ph.D. thesis, Stanford University, 1979.

[4] Low, J. R. **Automatic Coding: Choice of Data Structures.** ISR 16, Birkhaeuser Verlag, Basel, Switzerland, 1976.

[5] Morgenstern, M. *Automated Design and Optimization of Management Information System Software.* MIT Laboratory for Computer Science, Ph.D. thesis, September 1976.

[6] Rovner, P. D. *Automatic Representation Selection for Associative Data Structures.* Ph.D. thesis, Computer Science Department TR10, The University of Rochester, Rochester, New York, September 1976.

[7] Wegbreit, B. "Goal-Directed Program Transformation." in *Third ACM Symposium on Principles of Programming Languages*, January 1976.

[8] Schwartz, J. T. "Optimization of Very High Level Languages." in *Computer Languages*, Vol. 1, Permagon Press, Northern Ireland, 1975, 161-194.

[9] Rosenschein, S., and Katz, S. "Selection of Representations for Data Structures." In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, August, 1977.

[10] Rowe, L., and Tonge, F. M. "Automating the Selection of Implementation Structures". *IEEE Transactions on Software Engineering*, Vol. SE-4, 6, November 1978.